Sudoku Solver

Yixin Wang
wangyix@stanford.edu

***Abstract*--- An Android app was developed that allows a Sudoku puzzle to be extracted and solved in real time using captured images from the device's camera. The user sees a near real-time feed of the device camera. If the camera sees a Sudoku puzzle with reasonable clarity, the puzzle will be solved and the computed values will be overlaid on top of the camera feed with the proper perspective. While most of the steps of the implemented in this project are very similar to existing Sudoku solvers, the algorithm differs in how grid cell locations are determined. Typically, the Hough transform is used to detect the grid lines which are then used to partition the puzzle into cells. However, this is not robust to the case where the grid lines do not appear straight in the image, which occurs when the paper containing the puzzle is not lying flat. In this project, each grid cell is located individually using structuring elements to better handle cases where the rows and columns of cells are not perfectly straight.**

I. Introduction

Sudoku puzzles are a common sight in newspapers and puzzle books, and sometimes the solution is not available. While it is possible to manually type in the given numbers into a program or web app to solve the puzzle, it would be much faster and more convenient to use image processing to automatically detect and recognize the values in the puzzle. This app is designed to require minimal interaction from the user: simply aim the camera at the puzzle and the solution values automatically appear in the empty cells.

II. Previous Work

Many apps designed to accomplish the same thing already exist. A quick search in the Google Play Store reveals many such apps, like AR Sudoku Solver [1] and Sudoku Vision [2]. Luckily, one such app, Sudoku Grab on the App Store, had its algorithm explained in a blog post by the creator [3], and it works as follows. First, the input image is binarized using adaptive thresholding. A morphological open is applied to remove small specks of noise. Then, the largest connected component (blob) in the binary image is extracted. It's assumed that this blob contains the grid lines of the puzzle since the puzzle should be the main focus of the camera image. Next, the Hough transform is applied to the grid blob and the leftmost, rightmost, topmost, and bottommost lines are retrieved. These lines are assumed to be the outer boundaries of the puzzle. The four corners of the puzzle are calculated by intersecting the boundary lines. They are then used to calculate a homography to transform the binary puzzle image to fit a square image such that the grid lines are axis-aligned. The square image is divided evenly into 9x9 square cells, and the largest blob is extracted from each cell. The cells whose largest blob is above a certain threshold are assumed to be digits. They are then recognized using a neural network (which the author admitted is probably overkill). Finally, the puzzle solution is computed and the results are overlaid on the original camera image.

As mentioned, these steps are very common across Sudoku solvers. Another implementation described on the web is a guide by MathWorks on how to solve the puzzle from an image using MATLAB [4]. The steps are pretty much identical except two things: the cell division is done in the perspective of the original image (no homography is applied to correct for the perspective), and the digits are recognized using structuring elements instead of a neural net. One additional implementation described can be found at [5], where the cell division is done by detecting the internal grid lines using a Hough transform within the puzzle bounds instead of assuming a perfect 9x9 grid division.

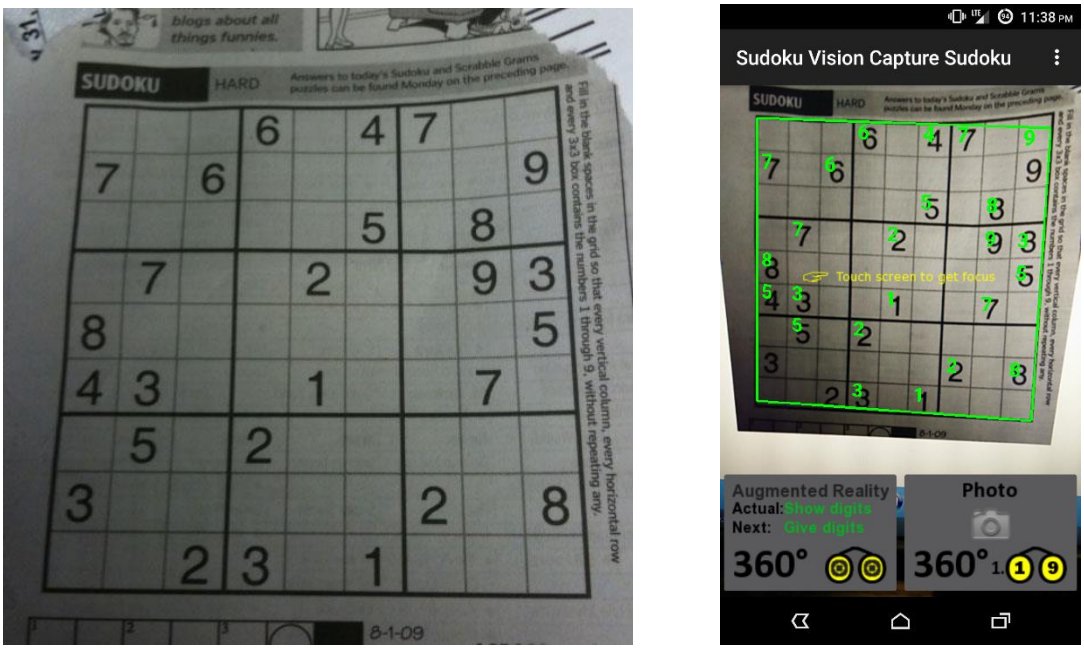

Figure 1. Left: a nonplanar puzzle. Right: screenshot from Sudoku Vision [2] when looking at the puzzle on the left

The potential problem with all these implementations is the assumption that the grid lines of the puzzle appear straight in the image, which is only true if the puzzle surface is planar in the real world when the image was taken (assuming no lens distortion). When a puzzle is nonplanar, dividing the puzzle into cells with straight lines may result in a bad partitioning of the puzzle, which will negatively affect the number recognition step. In figure 1 above, the "9" in the cell directly below the top right cell of the puzzle has been erroneously determined to be in the top right cell by the app. Note that the top right corner of the bounding quad shown by the app is much lower than the actual top right corner of the puzzle, which is probably the cause of the misplacement of the "9" since the app thought its location was much closer to the top right corner than it actually is. This motivates a better method of partitioning the cells in the case of nonplanar puzzles.

III. Overview of Algorithm

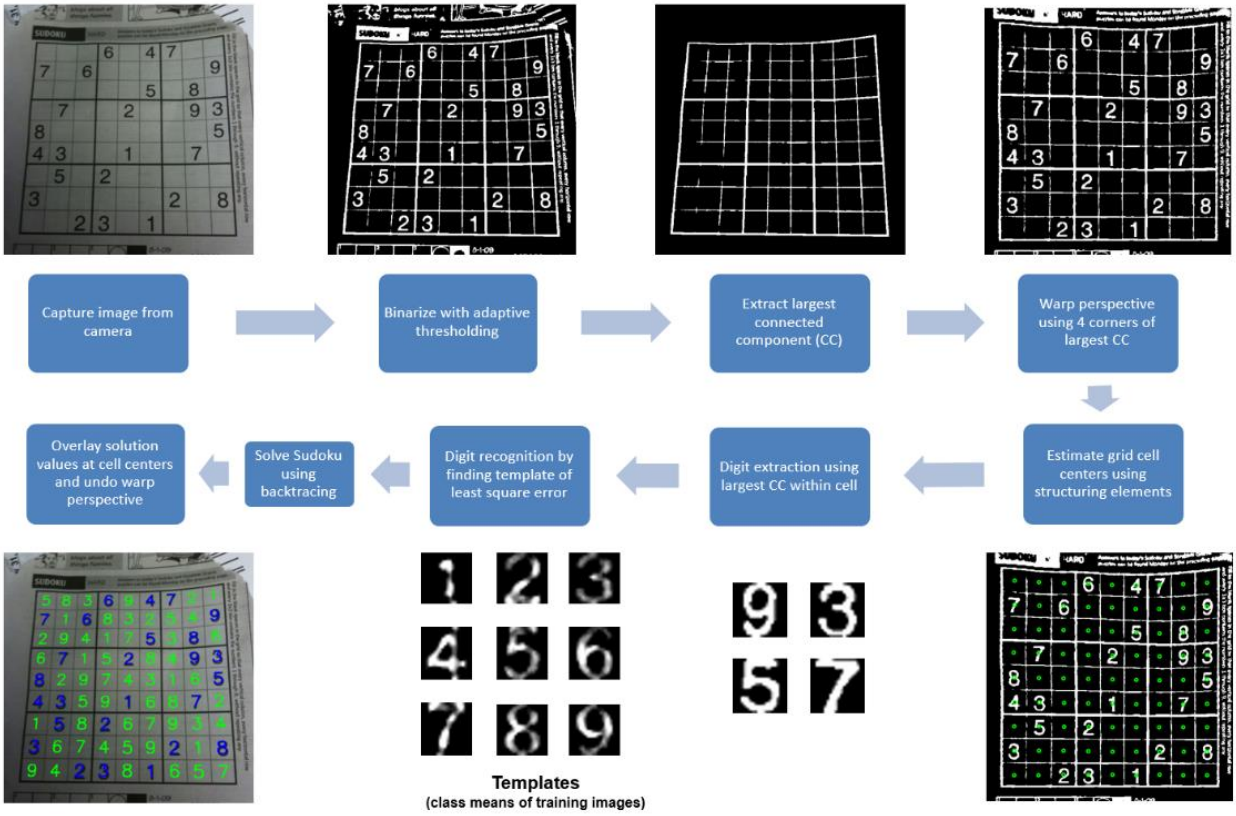

Figure 2. Major steps of the algorithm

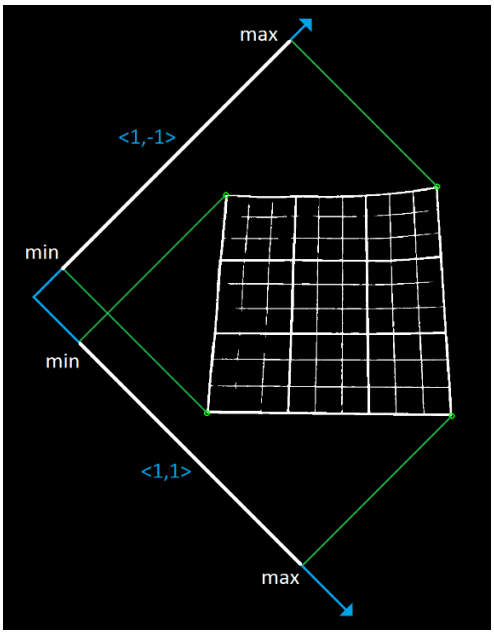

Figure 3. Finding four corners of grid by using projections

All the steps except the partitioning of grid cells are very similar to those of the implementations mentioned in the Previous Work section with a few small modifications and additional steps. The major steps are shown in figure 2. The captured image is binarized using adaptive thresholding, and the largest blob of the binary image is extracted and is assumed to be the Sudoku grid. The four corners of the grid are determined the following way: each foreground pixel in the grid blob will project its (x, y) coordinates onto the vectors $\langle 1,1 \rangle$ and $\langle 1, -1 \rangle$. The pixel with the minimum projections and the pixel with the maximum projection onto $\langle 1,1 \rangle$ are assumed to be the top-left and bottom-right corners of the grid, respectively. Similarly, the pixels with the min and max projections onto $\langle 1, -1 \rangle$ are assumed to be the bottom-left and the top-right corners of the grid, respectively. Figure 3 shows a visualization of this. Now using these four corners, a homography is calculated to transform the grid from the binary image to a square. A small border is left on all sides of the transformed grid in case the boundaries of the grid curve outwards. Next, the centers of each cell are individually estimated using structuring elements. For each cell, a square neighborhood of fixed size around each cell center is cropped out. The number of foreground pixels in this neighborhood is calculated; if it doesn't fall within a specified range, it's assumed that the cell does not contain a digit. If the number of cells that may contain a digit is less than 17 (the supposed minimum number of clues that a Sudoku must have to ensure a unique solution), the algorithm terminates early. This check is usually where the algorithm terminates when the camera image does not contain a Sudoku puzzle and instead contains some arbitrary scene. Now, for each viable cell, the maximum blob is extracted and is assumed to be the digit. The digit blob is cropped and scaled and compared to nine templates of the digits "1" through "9". The template producing the least square difference with the digit blob is selected as the recognized value. After all values are recognized, a sanity check is performed to make sure the rules of Sudoku have not been violated (i.e. no digit is repeated in a row, nor in a column, nor in any of the nine 3x3 subgrids). Then the Sudoku is solved using backtracing using the code found in [6]. Both the detected values and the solution values are written onto a blank image at the cell centers determined earlier, and the image is transformed back to the perspective of the original image. Finally, this image is blended into the original image to produce the output image displayed on the device screen.

A couple of the more interesting steps mentioned above will be described in more detail in the next sections.

IV. Cell Center Estimation

This step is where this approach differs from most existing Sudoku solvers. For nonplanar grids, dividing the grid evenly into 9x9 cells can negatively affect number recognition. In figure 4, a perfect 9x9 grid of squares is overlaid on top of the perspective-corrected Sudoku grid. While this particular input may not suffer much from partitioning the grid this way, it's easy to imagine if the grid was more warped, causing the partitioning to differ significantly from the actual grid lines. In such a case, if the grid is partitioned before digit blob extraction, many digits would be split in between different cells, producing a bad input for the recognition step. If the digit blobs are extracted without the grid being partitioned, it would still be difficult to tell which cell that digit belongs to. In either case, there should be a more accurate way to determine the cell locations.

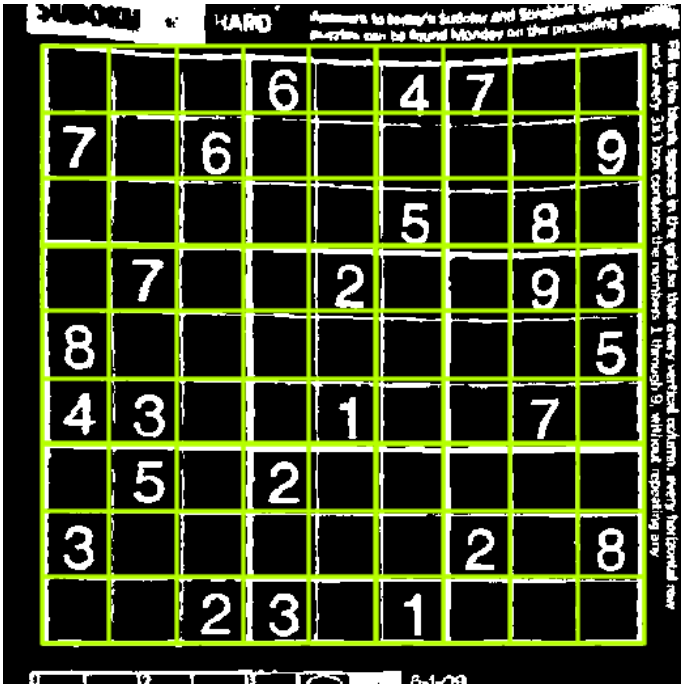

Figure 4. An even 9x9 partitioning of a nonplanar grid

The method implemented works as follows. We will start out with a rough estimate of the cell center locations. We do this by placing them at the centers of the partitions of an even 9x9 partitioning (i.e. place them at the center of the green cells in Figure 4). Then for each cell, we will refine its cell center position vertically, then horizontally.

Figure 5. Left: the two structuring elements used. Right: refining a cell center's vertical position using SE A

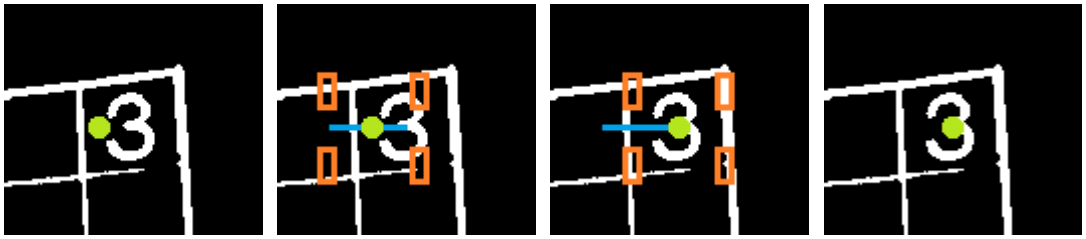

Figure 6.Refining a cell center's horizontal position, starting where vertical refinement left off. The final cell center estimate is shown in the far right image.

Figure 5 shows the two structuring elements we'll use. When a structuring element is centered at (x, y), its response is defined to be the sum of the values of the pixels within the four orange rectangles. SE A is used to refine the cell center vertically and SE B is used to refine the cell center horizontally. Both SEs are square and sized to be the same size as an evenly partitioned cell, or 1/81 of the area of the grid.

To vertically refine a cell center, the basic intuition is that we center SE A at the current cell center estimate and slide it up and down within some predefined range and place it where the highest number of foreground pixels will fall within the orange areas. The right three images of figure 5 visualize this process, where the green dot represents the cell center estimate and the blue line represents the range in which SE A's center can move. The rightmost image in figure 5 shows where the max response will occur: when the top and bottom grid lines of the cell coincide with the top two and bottom two orange areas of SE A. This might beg the question, why not just join the top two (or bottom two) orange areas of SE A into one long, thin area? Why have this gap in between? The answer is that this configuration reduces the effect that the number in the cell (or in adjacent cells) will have on the responses of the SEs. If a cell has a number, this gap allows that number to mostly pass through the SE without generating a high response. We're really only looking for the gridline pixels with these SEs, so we want the SE's orange areas to avoid the pixels of the number in the cell. After the cell center has been refined vertically, we do the same thing but transposed: we slide SE B left and right within a fixed range of the cell center estimate provided by vertical refinement and move the cell center horizontally to the location of highest response. This is shown in figure 6. There, the gap explanation is more easily seen: if SE B were just two long vertical bars instead of 4 shorter ones, it will pick up a strong response when the right vertical bar coincides with the pixels of the "3".

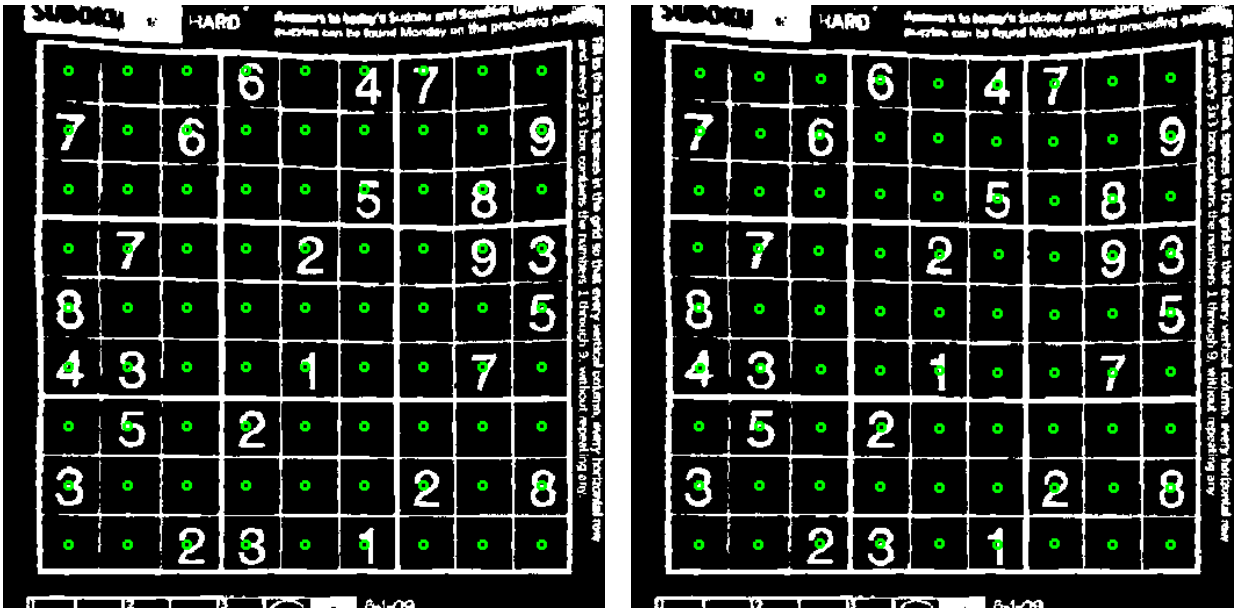

Figure 7. Left: initial cell center estimates based on even 9x9 division of square boundary. Right: cell center estimates after vertical and horizontal refinement.

V. Number Recognition

Nine templates of 20x20 pixels were used in the number recognition step. After the digit blob is extracted from the square neighborhood of each cell center, it's cropped, scaled (evenly in width and height), and centered in a 20x20 image. The templates were constructed from a training set of 289 images made exclusively of digit blobs produced by this algorithm. The template for a particular digit was set to be the difference between the mean of the training images of that digit (class mean) and the mean of all the training images. These nine templates are hard-coded into the app. During the recognition step, the mean of the extracted number images is calculated (test mean). To match a number image to a template, the test mean is first subtracted from it. Then, the mean square error (MSE) between that image and each of the template images is calculated. The template giving the smallest MSE is selected as the recognized value of that number.

Figure 8. The templates used, which are the class mean minus the overall mean of the training set.

VI. Results

Several methods were tried for digit recognition before settling on using least MSE. 370 20x20 images of digit blobs were extracted by this algorithm from many different images of Sudoku puzzles found online. Of those, 289 were used as training data and 81 were used as test data. Three different methods were tried: least MSE, eigenfaces, and fisherfaces. To classify a testing image, the calculated descriptor of the test image is used in a Euclidean nearest-neighbor search against the 9 class mean descriptors of the training data. The accuracy of classifying the testing images as well as the training images were determined. For eigenfaces, the accuracy seemed to peak when 50 eigenfaces were used, so that was used to generate the data below. For fisherfaces, 8 fisherfaces (generated from the 50-eigenfaces-space) was used. The accuracy of each of these three methods are shown in Table 1 below:

	Training Accuracy	Test Accuracy
Least MSE	99.65%	100%
50 Eigenfaces	99.65%	97.53%
8 Fisherfaces	100%	97.53%

Table 1. Accuracy of digit recognition of the three attempted methods.

As seen, the accuracy of all three methods are essentially equal. The slightly higher test accuracy of least MSE over the others is probably within the margin of error. From these results, least MSE was chosen as the method to implement in the app since it was by far the simplest and fastest method of the three with seemingly no reduction in accuracy.

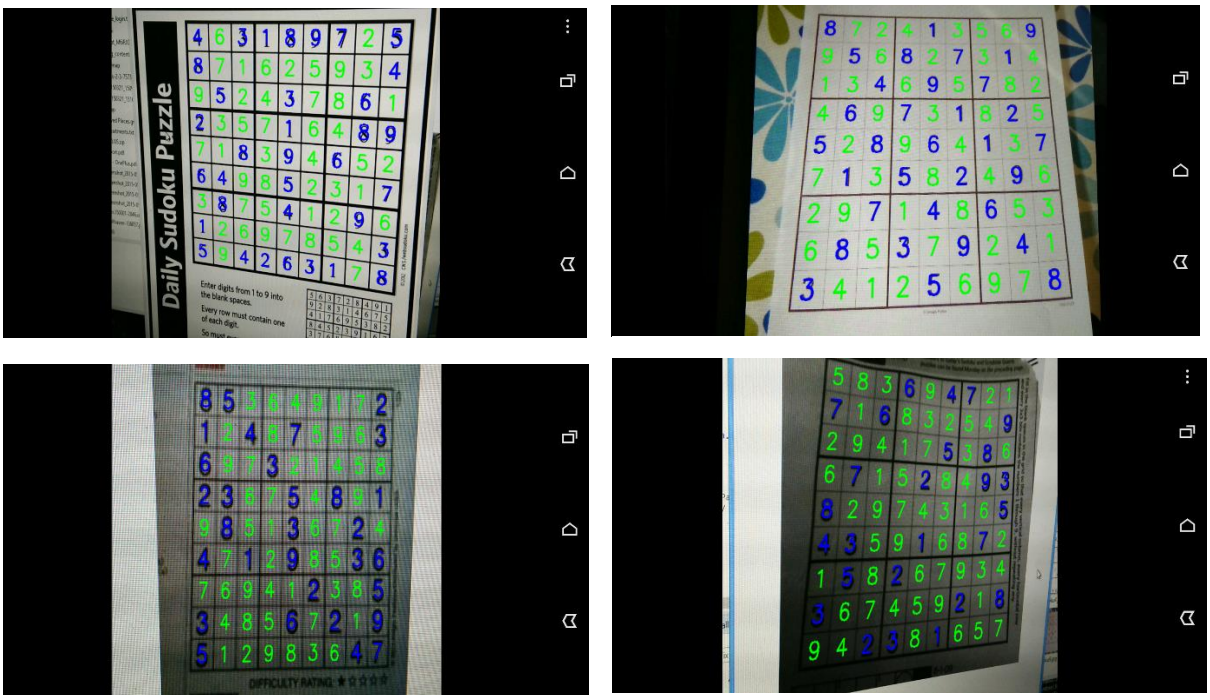

Figure 9. Some screenshots of the app taken while running.

Subjectively, the detection results in the final app were very good. It was very easy to get the solution to show up when used on a clean, noise-free image of a Sudoku with good contrast. Viewing slightly nonplanar Sudoku puzzles did not affect the detection results much; as long as the entire puzzle was in the frame of the camera and took up a large enough portion of the image, the algorithm worked.

On images with low contrast (or images with a few low-contrast areas), it was sometimes difficult to get the solution to show up consistently; the solution overlay would occasionally show an incorrect recognition of a number or two. Also, there was an unforeseen issue in the effect of motion blur and focus changes: if the device could not be held steady, the resulting motion blur could cause one of the algorithm steps to fail. Also, when the camera decides to refocus, the temporary blurring of the image will have the same negative effect. These effects made the app slightly less user-friendly to use.

From a performance standpoint, the app displayed around 5 to 10 frames per second on an Android smartphone with comparable specs to the Samsung Galaxy S5. While this is not a smooth framerate, the speed could still be considered real-time.

VII. Conclusion and Future Work

An algorithm for automatically solving Sudoku puzzles from mobile camera images was described. While many of the steps of the algorithm are very similar to existing Sudoku solvers, the grid partitioning step avoided an even partitioning in favor of using structuring elements to estimate each cell center individually. This change reduces the probability that a digit is broken up during partitioning or being erroneously assigned to a different cell than the one it actually

resides in. The final app was able to correctly extract digits from non-planar puzzles and puzzles viewed from an angle. Some possible improvements include:

A. Freezing the display once a solution has been successfully computed so that the user can comfortably view it without having to continue holding the camera still.

B. Improve the performance of the app by utilizing multithreading or GPU operations to improve the framerate for a smooth viewing experience. Another way could be to downsample the input image first to reduce computation time of the subsequence image processing steps.

C. Add the ability for the app to give out the solution step-by-step, showing the deduction of each new value using only the values that were known to the user at the time.

References

[1] Play.google.com, 'AR Sudoku Solver', 2015. [Online]. Available: https://play.google.com/store/apps/details?id=com.enigon.sudokusolver&hl=en. [Accessed: 31- May- 2015].

[2] Play.google.com, 'Sudoku Vision', 2015. [Online]. Available: https://play.google.com/store/apps/details?id=com.rogerlebo.sudokuvision&hl=en. [Accessed: 31- May- 2015].

[3] Sudokugrab.blogspot.com, 'iPhone Sudoku Grab: How does it all work?', 2009. [Online]. Available: http://sudokugrab.blogspot.com/2009/07/how-does-it-all-work.html. [Accessed: 31- May- 2015].

[4] T. Muppirala, 'Solving a Sudoku Puzzle Using a Webcam - MATLAB Video', *Mathworks.com*, 2011. [Online]. Available: http://www.mathworks.com/videos/solving-a-sudoku-puzzle-using-a-webcam-68773.html. [Accessed: 31- May- 2015].

[5] B. Banko, 'Realtime Webcam Sudoku Solver - CodeProject', *Codeproject.com*, 2011. [Online]. Available: http://www.codeproject.com/Articles/238114/Realtime-Webcam-Sudoku-Solver. [Accessed: 31- May- 2015].

[6] M. Bhojasia, 'C++ Program to Solve Sudoku Problem using BackTracking - Sanfoundry', *Sanfoundry*, 2013. [Online]. Available: http://www.sanfoundry.com/cpp-program-solve-sudoku-problem-backtracking/. [Accessed: 31- May- 2015].